FOUNDATION-0.10: Shared Utilities - PART 1 (Code)

🎯 CONTEXT

Phase: FOUNDATION (Week 1 - Day 4 Afternoon)
Component: Shared Python Utilities
Estimated Time: 15 min AI execution + 10 min verification
Complexity: LOW-MEDIUM
Risk Level: LOW
Files: Part 1 of 2 (Code implementation)
MILESTONE: Create reusable utilities for all Python agents! 🎯

📦 DEPENDENCIES

Must Complete First:

- **FOUNDATION-0.2a:** PostgreSQL Core Schema ✅ COMPLETED
- **FOUNDATION-0.3:** ClickHouse Schema ✅ COMPLETED
- **FOUNDATION-0.4:** Qdrant Setup ✅ COMPLETED
- **P-01:** Bootstrap Project ✅ COMPLETED

Required Services Running:

bash

```bash
# Verify base infrastructure
cd ~/optiinfra
docker ps
# Expected: PostgreSQL, Redis, ClickHouse, Qdrant running
```

🎯 OBJECTIVE

Build **Shared Python Utilities** that provide:

- ✅ Database connection management (PostgreSQL, ClickHouse, Qdrant, Redis)
- ✅ Configuration management (environment variables, settings)
- ✅ Logging utilities (structured logging, log levels)
- ✅ Common data models (base classes, validators)
- ✅ Time series helpers (data aggregation, time windows)
- ✅ Retry decorators (exponential backoff, error handling)
- ✅ Metric collectors (performance tracking)

Architecture:

```
┌──────────────────────────────────────────────────────┐
│          SHARED UTILITIES (Python)          │
│                              │
│  ┌────────────────────────────────────────────┐  │
│  │       Database Utilities          │  │
│  │  - PostgreSQL connection pool        │  │
│  │  - ClickHouse client             │  │
│  │  - Qdrant client               │  │
│  │  - Redis client               │  │
│  └────────────────────────────────────────────┘  │
│                              │
│  ┌────────────────────────────────────────────┐  │
│  │       Configuration Management       │  │
│  │  - Environment variables          │  │
│  │  - Settings validation           │  │
│  │  - Secrets management            │  │
│  └────────────────────────────────────────────┘  │
│                              │
│  ┌────────────────────────────────────────────┐  │
│  │       Logging & Monitoring          │  │
│  │  - Structured logging            │  │
│  │  - Performance metrics           │  │
│  │  - Error tracking              │  │
│  └────────────────────────────────────────────┘  │
│                              │
│  ┌────────────────────────────────────────────┐  │
│  │       Helper Functions           │  │
│  │  - Retry decorators             │  │
│  │  - Time series helpers           │  │
│  │  - Data validators             │  │
│  └────────────────────────────────────────────┘  │
└──────────────────────────────────────────────────────┘
          ↑
          │ Import
          │
┌──────────────────────────────────────────────────────┐
│          ALL AGENTS              │
│ from shared.database import get_postgres_conn      │
│ from shared.config import settings           │
```

```
from shared.logging import setup_logger
```

📁 FILE 1: Database Connection Manager

Location: `~/optiinfra/shared/database/connections.py`

python

```python
"""
Shared Database Connection Manager

Provides connection pooling and management for all databases.
"""

import os
import logging
from typing import Optional
from contextlib import contextmanager

import psycopg2
from psycopg2 import pool
from clickhouse_driver import Client as ClickHouseClient
from qdrant_client import QdrantClient
import redis

logger = logging.getLogger(__name__)


class DatabaseConnections:
    """Manages database connections with connection pooling"""

    def __init__(self):
        self._postgres_pool: Optional[pool.ThreadedConnectionPool] = None
        self._clickhouse_client: Optional[ClickHouseClient] = None
        self._qdrant_client: Optional[QdrantClient] = None
        self._redis_client: Optional[redis.Redis] = None

    def initialize_postgres(
        self,
        host: str = None,
        port: int = None,
        database: str = None,
        user: str = None,
        password: str = None,
        min_connections: int = 1,
        max_connections: int = 10,
    ):
        """Initialize PostgreSQL connection pool"""
        host = host or os.getenv('POSTGRES_HOST', 'localhost')
    port = port or int(os.getenv('POSTGRES_PORT', '5432'))
    database = database or os.getenv('POSTGRES_DB', 'optiinfra')
    user = user or os.getenv('POSTGRES_USER', 'optiinfra')
    password = password or os.getenv('POSTGRES_PASSWORD', 'password')

    try:
        self._postgres_pool = pool.ThreadedConnectionPool(
            minconn=min_connections,
            maxconn=max_connections,
            host=host,
            port=port,
            database=database,
            user=user,
            password=password,
        )
        logger.info(f"PostgreSQL connection pool initialized: {host}:{port}/{database}")
    except Exception as e:
        logger.error(f"Failed to initialize PostgreSQL pool: {e}")
        raise

def initialize_clickhouse(
    self,
    host: str = None,
    port: int = None,
    database: str = None,
    user: str = None,
    password: str = None,
):
    """Initialize ClickHouse client"""
    host = host or os.getenv('CLICKHOUSE_HOST', 'localhost')
    port = port or int(os.getenv('CLICKHOUSE_PORT', '8123'))
    database = database or os.getenv('CLICKHOUSE_DB', 'optiinfra')
    user = user or os.getenv('CLICKHOUSE_USER', 'default')
    password = password or os.getenv('CLICKHOUSE_PASSWORD', '')

    try:
        self._clickhouse_client = ClickHouseClient(
            host=host,
            port=port,
            database=database,
            user=user,
                password=password,
            )
            logger.info(f"ClickHouse client initialized: {host}:{port}/{database}")
        except Exception as e:
            logger.error(f"Failed to initialize ClickHouse client: {e}")
            raise

    def initialize_qdrant(
        self,
        host: str = None,
        port: int = None,
        api_key: str = None,
    ):
        """Initialize Qdrant client"""
        host = host or os.getenv('QDRANT_HOST', 'localhost')
        port = port or int(os.getenv('QDRANT_PORT', '6333'))
        api_key = api_key or os.getenv('QDRANT_API_KEY')

        try:
            self._qdrant_client = QdrantClient(
                host=host,
                port=port,
                api_key=api_key,
            )
            logger.info(f"Qdrant client initialized: {host}:{port}")
        except Exception as e:
            logger.error(f"Failed to initialize Qdrant client: {e}")
            raise

    def initialize_redis(
        self,
        host: str = None,
        port: int = None,
        db: int = None,
        password: str = None,
    ):
        """Initialize Redis client"""
        host = host or os.getenv('REDIS_HOST', 'localhost')
        port = port or int(os.getenv('REDIS_PORT', '6379'))
        db = db or int(os.getenv('REDIS_DB', '0'))
        password = password or os.getenv('REDIS_PASSWORD')
```

```python
        try:
            self._redis_client = redis.Redis(
                host=host,
                port=port,
                db=db,
                password=password,
                decode_responses=True,
            )
            # Test connection
            self._redis_client.ping()
            logger.info(f"Redis client initialized: {host}:{port}/{db}")
        except Exception as e:
            logger.error(f"Failed to initialize Redis client: {e}")
            raise

    @contextmanager
    def get_postgres_connection(self):
        """Get PostgreSQL connection from pool (context manager)"""
        if not self._postgres_pool:
            raise RuntimeError("PostgreSQL pool not initialized")

        conn = self._postgres_pool.getconn()
        try:
            yield conn
        finally:
            self._postgres_pool.putconn(conn)

    @contextmanager
    def get_postgres_cursor(self, commit: bool = True):
        """Get PostgreSQL cursor (context manager)"""
        with self.get_postgres_connection() as conn:
            cursor = conn.cursor()
            try:
                yield cursor
                if commit:
                    conn.commit()
            except Exception:
                conn.rollback()
                raise
            finally:
```

```
            cursor.close()

    @property
    def clickhouse(self) -> ClickHouseClient:
        """Get ClickHouse client"""
        if not self._clickhouse_client:
            raise RuntimeError("ClickHouse client not initialized")
        return self._clickhouse_client

    @property
    def qdrant(self) -> QdrantClient:
        """Get Qdrant client"""
        if not self._qdrant_client:
            raise RuntimeError("Qdrant client not initialized")
        return self._qdrant_client

    @property
    def redis(self) -> redis.Redis:
        """Get Redis client"""
        if not self._redis_client:
            raise RuntimeError("Redis client not initialized")
        return self._redis_client

    def close_all(self):
        """Close all database connections"""
        if self._postgres_pool:
            self._postgres_pool.closeall()
            logger.info("PostgreSQL pool closed")

        if self._clickhouse_client:
            self._clickhouse_client.disconnect()
            logger.info("ClickHouse client closed")

        if self._redis_client:
            self._redis_client.close()
            logger.info("Redis client closed")


# Global database connections instance
db_connections = DatabaseConnections()
```

```python
# Convenience functions
def get_postgres_connection():
    """Get PostgreSQL connection (context manager)"""
    return db_connections.get_postgres_connection()


def get_postgres_cursor(commit: bool = True):
    """Get PostgreSQL cursor (context manager)"""
    return db_connections.get_postgres_cursor(commit=commit)


def get_clickhouse_client() -> ClickHouseClient:
    """Get ClickHouse client"""
    return db_connections.clickhouse


def get_qdrant_client() -> QdrantClient:
    """Get Qdrant client"""
    return db_connections.qdrant


def get_redis_client() -> redis.Redis:
    """Get Redis client"""
    return db_connections.redis


def initialize_all_databases():
    """Initialize all database connections"""
    db_connections.initialize_postgres()
    db_connections.initialize_clickhouse()
    db_connections.initialize_qdrant()
    db_connections.initialize_redis()
    logger.info("All database connections initialized")
```

📁 FILE 2: Configuration Management

Location: `~/optiinfra/shared/config/settings.py`

python

```python
"""
Shared Configuration Management

Centralized configuration with environment variable support.
"""

import os
from typing import Optional
from dataclasses import dataclass


@dataclass
class DatabaseConfig:
    """Database configuration"""
    # PostgreSQL
    postgres_host: str = os.getenv('POSTGRES_HOST', 'localhost')
    postgres_port: int = int(os.getenv('POSTGRES_PORT', '5432'))
    postgres_db: str = os.getenv('POSTGRES_DB', 'optiinfra')
    postgres_user: str = os.getenv('POSTGRES_USER', 'optiinfra')
    postgres_password: str = os.getenv('POSTGRES_PASSWORD', 'password')

    # ClickHouse
    clickhouse_host: str = os.getenv('CLICKHOUSE_HOST', 'localhost')
    clickhouse_port: int = int(os.getenv('CLICKHOUSE_PORT', '8123'))
    clickhouse_db: str = os.getenv('CLICKHOUSE_DB', 'optiinfra')
    clickhouse_user: str = os.getenv('CLICKHOUSE_USER', 'default')
    clickhouse_password: str = os.getenv('CLICKHOUSE_PASSWORD', '')

    # Qdrant
    qdrant_host: str = os.getenv('QDRANT_HOST', 'localhost')
    qdrant_port: int = int(os.getenv('QDRANT_PORT', '6333'))
    qdrant_api_key: Optional[str] = os.getenv('QDRANT_API_KEY')

    # Redis
    redis_host: str = os.getenv('REDIS_HOST', 'localhost')
    redis_port: int = int(os.getenv('REDIS_PORT', '6379'))
    redis_db: int = int(os.getenv('REDIS_DB', '0'))
    redis_password: Optional[str] = os.getenv('REDIS_PASSWORD')

    @property
    def postgres_url(self) -> str:
        """Get PostgreSQL connection URL"""
        return f"postgresql://{self.postgres_user}:{self.postgres_password}@{self.postgres_host}:{self.postgres_port}/{self.


@dataclass
class OrchestratorConfig:
    """Orchestrator configuration"""
    host: str = os.getenv('ORCHESTRATOR_HOST', 'localhost')
    port: int = int(os.getenv('ORCHESTRATOR_PORT', '8080'))

    @property
    def url(self) -> str:
        """Get orchestrator URL"""
        return f"http://{self.host}:{self.port}"


@dataclass
class MockCloudConfig:
    """Mock Cloud Provider configuration"""
    host: str = os.getenv('MOCK_CLOUD_HOST', 'localhost')
    port: int = int(os.getenv('MOCK_CLOUD_PORT', '5000'))

    @property
    def url(self) -> str:
        """Get mock cloud URL"""
        return f"http://{self.host}:{self.port}"


@dataclass
class LoggingConfig:
    """Logging configuration"""
    level: str = os.getenv('LOG_LEVEL', 'INFO')
    format: str = os.getenv('LOG_FORMAT', 'json')  # 'json' or 'text'
    output: str = os.getenv('LOG_OUTPUT', 'stdout')  # 'stdout' or file path


@dataclass
class AgentConfig:
    """Agent configuration"""
    name: str = os.getenv('AGENT_NAME', 'agent')
    agent_type: str = os.getenv('AGENT_TYPE', 'generic')
    host: str = os.getenv('AGENT_HOST', 'localhost')
    port: int = int(os.getenv('AGENT_PORT', '8001'))
    version: str = os.getenv('AGENT_VERSION', '1.0.0')

    # Heartbeat settings
    heartbeat_interval: int = int(os.getenv('HEARTBEAT_INTERVAL', '30'))  # seconds

    # Task settings
    task_timeout: int = int(os.getenv('TASK_TIMEOUT', '300'))  # seconds
    max_concurrent_tasks: int = int(os.getenv('MAX_CONCURRENT_TASKS', '5'))


@dataclass
class Settings:
    """Global settings"""
    database: DatabaseConfig = DatabaseConfig()
    orchestrator: OrchestratorConfig = OrchestratorConfig()
    mock_cloud: MockCloudConfig = MockCloudConfig()
    logging: LoggingConfig = LoggingConfig()
    agent: AgentConfig = AgentConfig()

    # Environment
    environment: str = os.getenv('ENVIRONMENT', 'development')  # development, staging, production
    debug: bool = os.getenv('DEBUG', 'false').lower() == 'true'

    @property
    def is_production(self) -> bool:
        """Check if running in production"""
        return self.environment == 'production'

    @property
    def is_development(self) -> bool:
        """Check if running in development"""
        return self.environment == 'development'


# Global settings instance
settings = Settings()
```

📁 FILE 3: Logging Utilities

Location: `~/optiinfra/shared/logging/logger.py`

python

```python
"""
Shared Logging Utilities

Provides structured logging with JSON support.
"""

import logging
import sys
import json
from datetime import datetime
from typing import Any, Dict


class JSONFormatter(logging.Formatter):
    """JSON log formatter"""

    def format(self, record: logging.LogRecord) -> str:
        """Format log record as JSON"""
        log_data = {
            "timestamp": datetime.utcnow().isoformat(),
            "level": record.levelname,
            "logger": record.name,
            "message": record.getMessage(),
            "module": record.module,
            "function": record.funcName,
            "line": record.lineno,
        }

        # Add exception info if present
        if record.exc_info:
            log_data["exception"] = self.formatException(record.exc_info)

        # Add extra fields
        if hasattr(record, 'extra_fields'):
            log_data.update(record.extra_fields)

        return json.dumps(log_data)


class TextFormatter(logging.Formatter):
    """Human-readable text formatter"""
    def __init__(self):
        super().__init__(
            fmt='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
            datefmt='%Y-%m-%d %H:%M:%S'
        )


def setup_logger(
    name: str,
    level: str = 'INFO',
    format_type: str = 'text',
    output: str = 'stdout',
) -> logging.Logger:
    """
    Setup a logger with specified configuration.

    Args:
        name: Logger name
        level: Log level (DEBUG, INFO, WARNING, ERROR, CRITICAL)
        format_type: Format type ('json' or 'text')
        output: Output destination ('stdout', 'stderr', or file path)

    Returns:
        Configured logger
    """
    logger = logging.getLogger(name)
    logger.setLevel(getattr(logging, level.upper()))

    # Remove existing handlers
    logger.handlers = []

    # Create handler
    if output == 'stdout':
        handler = logging.StreamHandler(sys.stdout)
    elif output == 'stderr':
        handler = logging.StreamHandler(sys.stderr)
    else:
        handler = logging.FileHandler(output)

    # Set formatter
```

```python
    if format_type == 'json':
        formatter = JSONFormatter()
    else:
        formatter = TextFormatter()

    handler.setFormatter(formatter)
    logger.addHandler(handler)

    return logger


def log_with_context(logger: logging.Logger, level: str, message: str, **context):
    """
    Log message with additional context fields.

    Args:
        logger: Logger instance
        level: Log level
        message: Log message
        **context: Additional context fields
    """
    extra = {'extra_fields': context}
    log_method = getattr(logger, level.lower())
    log_method(message, extra=extra)
```

📁 FILE 4: Retry Decorator

Location: `~/optiinfra/shared/utils/retry.py`

python

```python
"""
Retry Decorator with Exponential Backoff

Provides automatic retry logic for functions.
"""

import time
import logging
from functools import wraps
from typing import Callable, Tuple, Type

logger = logging.getLogger(__name__)


def retry(
    max_attempts: int = 3,
    delay: float = 1.0,
    backoff: float = 2.0,
    exceptions: Tuple[Type[Exception], ...] = (Exception,),
    on_retry: Callable = None,
):
    """
    Retry decorator with exponential backoff.

    Args:
        max_attempts: Maximum number of retry attempts
        delay: Initial delay between retries (seconds)
        backoff: Backoff multiplier (delay *= backoff after each retry)
        exceptions: Tuple of exception types to catch
        on_retry: Optional callback function called on each retry

    Example:
        @retry(max_attempts=3, delay=1.0, backoff=2.0)
        def fetch_data():
            # This will retry up to 3 times with exponential backoff
            return requests.get(url)
    """
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            current_delay = delay
            last_exception = None

            for attempt in range(1, max_attempts + 1):
                try:
                    return func(*args, **kwargs)
                except exceptions as e:
                    last_exception = e

                    if attempt == max_attempts:
                        logger.error(
                            f"{func.__name__} failed after {max_attempts} attempts: {e}"
                        )
                        raise

                    logger.warning(
                        f"{func.__name__} failed (attempt {attempt}/{max_attempts}): {e}. "
                        f"Retrying in {current_delay}s..."
                    )

                    if on_retry:
                        on_retry(attempt, e)

                    time.sleep(current_delay)
                    current_delay *= backoff

            # This should never be reached, but just in case
            raise last_exception

        return wrapper
    return decorator


def async_retry(
    max_attempts: int = 3,
    delay: float = 1.0,
    backoff: float = 2.0,
    exceptions: Tuple[Type[Exception], ...] = (Exception,),
):
    """
    Async retry decorator with exponential backoff.
```

```
Example:
    @async_retry(max_attempts=3)
    async def fetch_data():
        async with aiohttp.ClientSession() as session:
            async with session.get(url) as response:
                return await response.json()
"""
def decorator(func):
    @wraps(func)
    async def wrapper(*args, **kwargs):
        import asyncio

        current_delay = delay
        last_exception = None

        for attempt in range(1, max_attempts + 1):
            try:
                return await func(*args, **kwargs)
            except exceptions as e:
                last_exception = e

                if attempt == max_attempts:
                    logger.error(
                        f"{func.__name__} failed after {max_attempts} attempts: {e}"
                    )
                    raise

                logger.warning(
                    f"{func.__name__} failed (attempt {attempt}/{max_attempts}): {e}. "
                    f"Retrying in {current_delay}s..."
                )

                await asyncio.sleep(current_delay)
                current_delay *= backoff

        raise last_exception

    return wrapper
return decorator
```

📁 FILE 5: Time Series Helpers

Location: `~/optiinfra/shared/utils/timeseries.py`

python

```python
"""
Time Series Helper Functions

Utilities for working with time series data.
"""

from datetime import datetime, timedelta
from typing import List, Dict, Tuple, Optional
import statistics


def create_time_windows(
    start_time: datetime,
    end_time: datetime,
    window_size: timedelta,
) -> List[Tuple[datetime, datetime]]:
    """
    Create time windows between start and end time.

    Args:
        start_time: Start datetime
        end_time: End datetime
        window_size: Size of each window

    Returns:
        List of (window_start, window_end) tuples

    Example:
        windows = create_time_windows(
            start_time=datetime(2025, 1, 1),
            end_time=datetime(2025, 1, 2),
            window_size=timedelta(hours=1)
        )
        # Returns 24 hourly windows
    """
    windows = []
    current = start_time

    while current < end_time:
        window_end = min(current + window_size, end_time)
        windows.append((current, window_end))
        current = window_end

    return windows


def aggregate_time_series(
    data: List[Dict],
    timestamp_field: str,
    value_field: str,
    window_size: timedelta,
    aggregation: str = 'avg',
) -> List[Dict]:
    """
    Aggregate time series data into windows.

    Args:
        data: List of data points with timestamps
        timestamp_field: Name of timestamp field
        value_field: Name of value field to aggregate
        window_size: Size of aggregation window
        aggregation: Aggregation function ('avg', 'sum', 'min', 'max', 'count')

    Returns:
        List of aggregated data points

    Example:
        data = [
            {'timestamp': datetime(2025, 1, 1, 0, 0), 'cpu': 45.2},
            {'timestamp': datetime(2025, 1, 1, 0, 5), 'cpu': 52.1},
            ...
        ]

        result = aggregate_time_series(
            data=data,
            timestamp_field='timestamp',
            value_field='cpu',
            window_size=timedelta(hours=1),
            aggregation='avg'
        )
    """
    if not data:
        return []

    # Sort by timestamp
    sorted_data = sorted(data, key=lambda x: x[timestamp_field])

    # Get time range
    start_time = sorted_data[0][timestamp_field]
    end_time = sorted_data[-1][timestamp_field]

    # Create windows
    windows = create_time_windows(start_time, end_time, window_size)

    # Aggregate data
    aggregated = []

    for window_start, window_end in windows:
        # Filter data in this window
        window_data = [
            d[value_field]
            for d in sorted_data
            if window_start <= d[timestamp_field] < window_end
        ]

        if not window_data:
            continue

        # Calculate aggregation
        if aggregation == 'avg':
            value = statistics.mean(window_data)
        elif aggregation == 'sum':
            value = sum(window_data)
        elif aggregation == 'min':
            value = min(window_data)
        elif aggregation == 'max':
            value = max(window_data)
        elif aggregation == 'count':
            value = len(window_data)
        else:
            raise ValueError(f"Unknown aggregation: {aggregation}")

        aggregated.append({
            'window_start': window_start,
            'window_end': window_end,
            'value': value,
            'count': len(window_data),
        })

    return aggregated


def calculate_rate_of_change(
    data: List[Dict],
    timestamp_field: str,
    value_field: str,
) -> List[Dict]:
    """
    Calculate rate of change between consecutive data points.

    Args:
        data: List of data points
        timestamp_field: Name of timestamp field
        value_field: Name of value field

    Returns:
        List of data points with rate_of_change field
    """
    if len(data) < 2:
        return []

    sorted_data = sorted(data, key=lambda x: x[timestamp_field])
    result = []

    for i in range(1, len(sorted_data)):
        prev = sorted_data[i - 1]
        curr = sorted_data[i]

        time_diff = (curr[timestamp_field] - prev[timestamp_field]).total_seconds()
        value_diff = curr[value_field] - prev[value_field]

        if time_diff > 0:
            rate = value_diff / time_diff
        else:
            rate = 0

        result.append({
            'timestamp': curr[timestamp_field],
            'value': curr[value_field],
            'rate_of_change': rate,
        })

    return result


def detect_anomalies(
    data: List[float],
    threshold_std_dev: float = 2.0,
) -> List[int]:
    """
    Detect anomalies using standard deviation method.

    Args:
        data: List of numeric values
        threshold_std_dev: Number of standard deviations for anomaly threshold

    Returns:
        List of indices where anomalies were detected

    Example:
        values = [10, 12, 11, 10, 50, 11, 12]  # 50 is an anomaly
        anomaly_indices = detect_anomalies(values)
        # Returns [4]
    """
    if len(data) < 3:
        return []

    mean = statistics.mean(data)
    std_dev = statistics.stdev(data)

    anomalies = []
    for i, value in enumerate(data):
        if abs(value - mean) > threshold_std_dev * std_dev:
            anomalies.append(i)
    return anomalies
```

📁 FILE 6: Data Validators

Location: `~/optiinfra/shared/utils/validators.py`

python

```
"""
Data Validation Utilities

Common validators for data validation.
"""

import re
from typing import Any, List, Optional
from datetime import datetime


class ValidationError(Exception):
    """Validation error exception"""
    pass


def validate_required(value: Any, field_name: str):
    """Validate that a value is not None or empty"""
    if value is None or (isinstance(value, str) and not value.strip()):
        raise ValidationError(f"{field_name} is required")


def validate_email(email: str) -> bool:
    """Validate email format"""
    pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z
```